CAPÍTULO 6

ARRAYS NUMÉRICOS: VECTORES Y MATRICES

Hasta el momento hemos trabajado con variables, declaradas una a una en la medida en que nos han sido necesarias. Pero pudiera ocurrir que necesitásemos un bloque de variables grande, por ejemplo para definir los valores de una matriz numérica, o para almacenar los distintos valores obtenidos en un proceso de cálculo del que se obtienen numerosos resultados del mismo tipo. Supongamos, por ejemplo, que deseamos hacer un programa que ordene mil valores enteros: habrá que declarar entonces mil variables, todas ellas del mismo tipo, y todas ellas con un nombre diferente.

Es posible hacer una declaración conjunta de un grupo de variables. Eso se realiza cuando todas esas variables son del mismo tipo. Esa es, intuitivamente, la noción del array. Y ese es el concepto que introducimos en el presente capítulo

Noción y declaración de array.

Un **array** (también llamado **vector**) es una colección de variables del mismo tipo todas ellas referenciadas con un nombre común.

La sintaxis para la declaración de un vector es la siguiente:

tipo nombre_vector[dimensión];

Donde *tipo* define el tipo de dato de todas las variables creadas, y *dimensión* es un literal que indica cuántas variables de ese *tipo* se deben crear. En ningún caso está permitido introducir el valor de la dimensión mediante una variable. El compilador reserva el espacio necesario para almacenar, de forma contigua, tantas variables como indique el literal *dimensión*: reservará, pues, tantos bytes como requiera una de esas variables, multiplicado por el número de variables a crear.

Por ejemplo, la sentencia **short int** `mi_vector[1000];` reserva dos mil bytes de memoria consecutivos para poder codificar mil variables de tipo ***short***.

Cada una de las variables de un array tiene un comportamiento completamente independiente de las demás. Su única relación con todas las otras variables del array es que están situadas todas ellas de forma correlativa en la memoria. Cada variable tiene su propio modo de ser llamada: desde *nombre_vector[0]* hasta *nombre_vector[dimensión – 1]*. En el ejemplo anterior, tendremos 1000 variables que van desde *mi_vector[0]* hasta *mi_vector[999]*.

C no comprueba los límites del vector. Es responsabilidad del programador asegurar que no se accede a otras posiciones de memoria contiguas del vector. Por ejemplo, si hacemos referencia al elemento *mi_vector[1000]*, el compilador no dará como erróneo ese nombre, aunque de hecho no exista tal variable.

La variable *mi_vector[0]* está posicionada en una dirección de memoria cualquiera. La variable *mi_vector[1]* está situada dos bytes más adelante, porque *mi_vector* es un array de tipo ***short*** y por tanto *mi_vector[0]* ocupa 2 bytes (como todos los demás elementos del vector), y porque *mi_vector[1]* es consecutiva en la memoria, a *mi_vector[0]*. Esa sucesión de ubicaciones sigue en adelante, y la variable *mi_vector[999]* estará 1998 bytes por encima de la posición de *mi_vector[0]*. Si hacemos referencia a la variable *mi_vector[1000]* entonces el compilador considera la posición de memoria situada 2000 bytes por encima de la posición de *mi_vector[0]*. Y de allí tomará valor o escribirá valor si así se lo indicamos. Pero realmente, en esa posición el ordenador no tiene reservado espacio para esta variable, y no sabemos qué estaremos realmente leyendo o modificando. Este tipo de errores son muy graves y a veces no se detectan hasta después de varias ejecuciones.

El recorrido del vector se puede hacer mediante índices. Por ejemplo:

```
short mi_vector[1000], i;
for(i = 0 ; i < 1000 ; i++) mi_vector[i] = 0;
```

Este código recorre todo el vector e inicializa a cero todas y cada una de sus variables.

Téngase cuidado, por ejemplo, con el recorrido del vector, que va desde el elemento 0 hasta el elemento dimensión – 1. Un error habitual es escribir el siguiente código:

```
short mi_vector[1000], i;
for(i = 0 ; i <= 1000 ; i++) mi_vector[i] = 0;
```

Donde se hará referencia a la posición 1000 del vector, que no es válida.

Existe otro modo de inicializar los valores de un vector o array, sin necesidad de recorrerlo con un índice. Se puede emplear para ello el operador asignación, dando, entre llaves y separados por comas, tantos valores como dimensión tenga el vector. Por ejemplo;

```
short mi_vector[10] = {10,20,30,40,50,60,70,80,90,100};
```

que es equivalente al siguiente código:

```
short mi_vector[10], i;
for(i = 0 ; i < 10 ; i++) mi_vector[i] = 10 * (i + 1);
```

Cuando se inicializa un vector mediante el operador asignación en su declaración, como hay que introducir entre llaves tantos valores como sea la dimensión del vector creado, es redundante indicar la dimensión entre corchetes y también en el cardinal del conjunto de valores asignado. Por eso, se puede declarar ese vector sin especificar el número de variables que se deben crear. Por ejemplo:

```
short mi_vector[] = {10,20,30,40,50,60,70,80,90,100};
```

Por lo demás, estas variables son exactamente iguales que todas las vistas hasta el momento. También ellas pueden ser declaradas globales o locales, o ***static***, o ***extern***.

Noción y declaración de array de dimensión múltiple, o matrices.

Es posible definir arrays de más de una dimensión. El comportamiento de esas variables vuelve a ser conceptualmente muy sencillo. La sintaxis de esa declaración es la siguiente:

tipo nombre_matriz[dim_1][dim_2]…[dim_N];

Donde los valores de las dimensiones son todos ellos literales.

Por ejemplo podemos crear una matriz tres por tres:

```
float matriz[3][3];
```

que reserva 9 bloques de cuatro bytes cada uno para poder almacenar valores tipo ***float***. Esas variables se llaman también con índices, en este caso dos índices (uno para cada dimensión) que van desde el 0 hasta el valor de cada dimensión menos uno.

Por ejemplo:

```
long matriz[5][2], i, j;
```

```
for(i = 0 ; i < 5 ; i++)
      for(j = 0 ; j < 2 ; j++)
            matriz[i][j] = 0;
```

donde tenemos una matriz de cinco filas y dos columnas, toda ella con los valores iniciales a cero. También se puede inicializar la matriz mediante el operador asignación y llaves. En este caso se haría lo siguiente:

```
long int matriz[5][2] = {{1,2},{3,4},{5,6},{7,8},{9,10}};
```

que es lo mismo que escribir

```
long matriz[5][2], i, j, k;
for(i = 0 , k = 1; i < 5 ; i++)
      for(j = 0 ; j < 2 ; j++)
      {
            matriz[i][j] = k;
            k++;
      }
```

Y de nuevo hay que estar muy vigilante para no sobrepasar, al utilizar los índices, la dimensión de la matriz. Para comprender mejor cómo se distribuyen las variables en la memoria, y el peligro de equivocarse en los índices que recorren la matriz, veamos el siguiente programa que crea una matriz de 2 por 5 y muestra por pantalla la dirección de cada uno de los 10 elementos:

```
#include <stdio.h>
void main(void)
{
      char m[2][5];
      short i, j;
      for(i = 0 ; i < 2 ; i++)
            for(j = 0 ; j < 5 ; j++)
                  printf("&m[%hd][%hd] = %p\n",i,j,&m[i][j]);
}
```

La salida por pantalla ha sido esta:

```
&m[0][0] = 0012FF80
&m[0][1] = 0012FF81
&m[0][2] = 0012FF82
&m[0][3] = 0012FF83
&m[0][4] = 0012FF84
&m[1][0] = 0012FF85
&m[1][1] = 0012FF86
```

```
&m[1][2] = 0012FF87
&m[1][3] = 0012FF88
&m[1][4] = 0012FF89
```

Donde vemos que los elementos van ordenados desde el *m[0][0]* hasta el *m[0][4]*, y a continuación el *m[1][0]*: cuando termina la primera fila comienza la segunda fila.

Si ahora, por equivocación, escribiéramos el siguiente código

```
#include <stdio.h>
void main(void)
{
      char m[2][5];
      short i, j;
      for(i = 0 ; i < 2 ; i++)
            for(j = 0 ; j < 6 ; j++)
                  printf("&m[%hd][%hd] = %p\n",i,j,&m[i][j]);
}
```

(donde, como se ve, el índice j recorre hasta el valor 5, y no sólo hasta el valor 4) tendríamos la siguiente salida por pantalla:

```
&m[0][0] = 0012FF80
&m[0][1] = 0012FF81
&m[0][2] = 0012FF82
&m[0][3] = 0012FF83
&m[0][4] = 0012FF84
&m[0][5] = 0012FF85
&m[1][0] = 0012FF85
&m[1][1] = 0012FF86
&m[1][2] = 0012FF87
&m[1][3] = 0012FF88
&m[1][4] = 0012FF89
&m[1][5] = 0012FF8A
```

Donde el compilador “se ha tragado” que la matriz tiene el elemento *m[0][5]* y el *m[1][5]*. No sabemos qué habrá en la posición de la variable no existente *m[1][5]*. Sí sabemos en cambio qué hay en la *m[0][5]*: si vemos la lista de la salida, tenemos que el compilador considera que la variable *m[0][5]* estará a continuación de la *m[0][4]*. Pero por otro lado, ella sabe que la segunda fila comienza en *m[1][0]* y sabe dónde está ubicada. Si comparamos *&m[0][5]* y *&m[1][0]* veremos que a ambos se les supone la misma dirección. Y es que *m[0][5]* no ocupa lugar porque no existe. Pero cuando se escriba en el código

m[0][5] = 0;

lo que estaremos poniendo a cero, quizá sin saberlo, es la variable *m[1][0]*.

Conclusión: mucho cuidado con los índices al recorrer matrices y vectores.

Ejercicios

41. ***Escriba el código necesario para crear una matriz identidad (todos sus valores a cero, excepto la diagonal principal) de dimensión 3.***

```
short identidad[3][3] = {{1,0,0},{0,1,0},{0,0,1}};
```

Otra forma de solventarlo:

```
short identidad[3][3], i, j;
for(i = 0 ; i < 3 ; i++)
      for(j = 0 ; j < 3 ; j++)
            identidad[i][j] = i == j ? 1 : 0;
```

El operador ?: se presentó al hablar de las estructuras de control condicionales. Como el lenguaje C devuelve el valor 1 cuando una expresión se evalúa como verdadera, hubiera bastando con que la última línea del código presentado fuese

```
            identidad[i][j] = i == j;
```

Para mostrar la matriz por pantalla el código es siempre más o menos el mismo:

```
for(i = 0 ; i < 3 ; i++)
{
      printf("\n\n");
      for(j = 0 ; j < 3 ; j++)
            printf("%5hd",identidad[i][j]);
}
```

42. ***Escriba un programa que solicite al usuario los valores de una matriz de tres por tres y muestre por pantalla la traspuesta de esa matriz introducida.***

```
#include <stdio.h>
void main(void)
{
	short matriz[3][3];
	short i, j;
	for(i = 0 ; i < 3 ; i++)
		for(j = 0 ; j < 3 ; j++)
		{
			printf("matriz[%hd][%hd] = ", i, j);
			scanf("%hd",&matriz[i][j]);
		}
	for(i = 0 ; i < 3 ; i++)
	{
		printf("\n\n");
		for(j = 0 ; j < 3 ; j++)
			printf("%5hd",matriz[i][j]);
	}
	printf("\n\n\n");
	for(i = 0 ; i < 3 ; i++)
	{
		printf("\n\n");
		for(j = 0 ; j < 3 ; j++)
			printf("%5hd",matriz[j][i]);
	}
}
```

Primero muestra la matriz tal y como la ha introducido el usuario, y más abajo muestra su traspuesta.

43. ***Escriba un programa que solicite al usuario los valores de dos matrices de tres por tres y muestre por pantalla cada una de ellas, una al lado de la otra, y su suma, y cada una de ellas, una al lado de la otra, y su producto.***

```
#define TAM 3
#include <stdio.h>
void main(void)
{
      short a[TAM][TAM];
      short b[TAM][TAM];
      short s[TAM][TAM];
      short p[TAM][TAM];
      short i, j, k;
// Entrada matriz a.
      for(i = 0 ; i < TAM ; i++)
            for(j = 0 ; j < TAM ; j++)
            {
                  printf("a[%hd][%hd] = ", i, j);
                  scanf("%hd",&a[i][j]);
            }
// Entrada matriz b.
      for(i = 0 ; i < TAM ; i++)
            for(j = 0 ; j < TAM ; j++)
            {
                  printf("b[%hd][%hd] = ", i, j);
                  scanf("%hd",&b[i][j]);
            }
// Cálculo Suma.
      for(i = 0 ; i < TAM ; i++)
            for(j = 0 ; j < TAM ; j++)
                  s[i][j] = a[i][j] + b[i][j];
// Cálculo Producto.
// p[i][j]=a[i][0]*b[0][j]+a[i][1]*b[1][j]+a[i][2]*b[2][j]
      for(i = 0 ; i < TAM ; i++)
            for(j = 0 ; j < TAM ; j++)
            {
                  p[i][j] = 0;
                  for(k = 0 ; k < TAM ; k++)
                        p[i][j] += a[i][k] * b[k][j];
            }
// Mostrar resultados.
// SUMA
      for(i = 0 ; i < TAM ; i++)
      {
            printf("\n\n");
            for(j = 0 ; j < TAM ; j++)
                  printf("%4hd",a[i][j]);
            printf("\t");
            for(j = 0 ; j < TAM ; j++)
                  printf("%4hd",b[i][j]);
            printf("\t");
            for(j = 0 ; j < TAM ; j++)
                  printf("%4hd",s[i][j]);
            printf("\t");
      }
```

```
// PRODUCTO
      printf("\n\n\n");
      for(i = 0 ; i < TAM ; i++)
      {
            printf("\n\n");
            for(j = 0 ; j < TAM ; j++)
                  printf("%4hd",a[i][j]);
            printf("\t");
            for(j = 0 ; j < TAM ; j++)
                  printf("%4hd",b[i][j]);
            printf("\t");
            for(j = 0 ; j < TAM ; j++)
                  printf("%4hd",p[i][j]);
            printf("\t");
      }
}
```

En el manejo de matrices y vectores es frecuente utilizar siempre, como estructura de control de iteración, la opción ***for***. Y es que tiene una sintaxis que permite manipular muy bien las iteraciones que tienen un número prefijado de repeticiones. Más, en el caso de las matrices, que las mismas variables de control de la estructura son las que sirven para los índices que recorre la matriz.

44. ***Escriba un programa que solicite al usuario un conjunto de valores (tantos como quiera el usuario) y que al final, ordene esos valores de menor a mayor. El usuario termina su entrada de datos cuando introduzca el cero.***

```
#include <stdio.h>
void main(void)
{
      short datos[1000];
      short i, j, nn;
// Introducción de datos.
      i = 0;
      do
      {
            printf("Entada de nuevo dato ... ");
            scanf("%hi",&datos[i]);
            i++;
```

```
      }while(datos[i - 1] != 0 && i < 1000);
      nn = i - 1; // Total de datos válidos introducidos.
// Ordenar datos
      for(i = 0 ; i <= nn ; i++)
            for(j = i + 1 ; j < nn ; j++)
                  if(datos[i] > datos[j])
                  {
                        datos[i] ^= datos[j];
                        datos[j] ^= datos[i];
                        datos[i] ^= datos[j];
                  }
// Mostrar datos ordenados por pantalla
      printf("\n\n");
      for(i = 0 ; i < nn ; i++)
            printf("%li < ", datos[i]);
      printf("\b\b  ");
}
```

Introducción de datos: va solicitando uno a uno todos los datos, mediante una estructura de control ***do–while***. La entrada de datos termina cuando la condición es falsa: o cuando se haya introducido un cero o cuando se hayan introducido tantos valores como enteros se han creado en el vector. Se habrán introducido tantos datos como indique el valor de la variable *i*, donde hay que tener en cuenta que ha sufrido un incremento también cuando se ha introducido el cero, y ese último valor no nos interesa. Por eso ponemos la variable *nn* al valor *i – 1*.

Ordenar datos: Tiene una forma parecida a la que se presentó para la ordenación de cuatro enteros (en el tema de las estructuras de control), pero ahora para una cantidad desconocida para el programador (recogida en la variable *nn*). Por eso se deben recorrer todos los valores mediante una estructura de iteración, y no como en el ejemplo de los cuatro valores que además no estaban almacenados en vectores, y por lo tanto no se podía recorrer los distintos valores mediante índices. Los datos quedan almacenados en el propio vector, de menor a mayor.

Mostrar datos: Se va recorriendo el vector desde el principio hasta el valor *nn*: esos son los elementos del vector que almacenan datos introducidos por el usuario.

45. ***Escribir un programa que solicite al usuario un entero positivo e indique si ese número introducido es primo o compuesto. Además, si el número es compuesto, deberá guardar todos sus divisores y mostrarlos por pantalla.***

```
#include <stdio.h>
#define TAM 1000
void main(void)
{
      unsigned long int numero, mitad;
      unsigned long int i;
      unsigned long int div;
      unsigned long int D[TAM];
      for(i = 0 ; i < TAM ; i++) D[i] = 0;
      D[0] = 1;
      printf("Numero que vamos a testear ... ");
      scanf("%lu", &numero);
      mitad = numero / 2;
      for(i = 1 , div = 2 ; div <= mitad ; div++)
      {
            if(numero % div == 0)
            {
                  D[i] = div;
                  i++;
                  if(i == TAM)
                  {
                        printf("Vector mal dimensionado.");
                        break;
                  }
            }
      }
      if(i < TAM) D[i] = numero;
      if(i == 1) printf("\n%lu es PRIMO.\n",numero);
      else
      {
            printf("\n%lu es COMPUESTO. ", numero);
            printf("Sus divisores son:\n\n");
            for(i = 0 ; i < TAM && D[i] != 0; i++)
                  printf("\n%lu", D[i]);
      }
}
```

Este programa es semejante a uno presentado en el capítulo de las estructuras de control. Allí se comenta el diseño de este código. Ahora añadimos que, cada vez que se encuentra un divisor, se almacena en

una posición del vector *D* y se incrementa el índice del vector (variable *i*). Se inicia el contador al valor 1 porque a la posición 0 del vector ya se le ha asignado el valor 1.

La variable *i* hace de centinela y de chivato. Si después de buscar todos los divisores la variable *i* está al valor 1, entonces es señal de que no se ha encontrado ningún divisor distinto del 1 y del mismo número, y por tanto ese número es primo.

Para la dimensión del vector se utiliza una constante definida con la directiva de procesador ***define***. Si se desea cambiar ese valor, no será necesario revisar todo el código en busca de las referencias a los límites de la matriz, sino que todo el código está ya escrito sobre ese valor prefijado. Basta cambiar el valor definido en la directiva para que se modifiquen todas las referencias al tamaño del vector.

46. ***Escribir un programa que defina un array de short de 32 elementos, y que almacene en cada uno de ellos los sucesivos dígitos binarios de un entero largo introducido por pantalla. Luego, una vez obtenidos todos los dígitos, el programa mostrará esos dígitos.***

Un posible código que da solución a este programa podría er el siguiente:

```
#include <stdio.h>

void main(void)
{
      signed long N;
      unsigned short bits[32], i;
      unsigned long Test;

      do
      {
            printf("\n\nIntroduce un entero ... ");
```

```
            scanf("%li",&N);

            if(N == 0) break;

            for(i=0,Test = 0x80000000 ; Test ; Test >>=1,i++)
                  bits[i] = Test & N ? 1 : 0;

            printf("\nCodificación binaria interna ... ");
            for(i = 0 ; i < sizeof(long) * 8 ; i++)
                  printf("%hu", bits[i]);

      }while(1);
}
```

Este código permite introducir tantos enteros como quiera el usuario. Cuando el usuario introduzca el valor cero entonces se termina la ejecución del programa. Ya quedó explicado el funcionamiento de este algoritmo en un tema anterior. Ahora simplemente hemos introducido la posibilidad de que se almacenen los dígitos binarios en un array.

Una posible salida por pantalla de este programa sería la siguiente:

Introduce un entero ... 12
Codificación binaria interna ... 00000000000000000000000000001100
Introduce un entero ... -12
Codificación binaria interna ... 11111111111111111111111111110100
Introduce un entero ... 0

47. ***Un cuadro mágico es un reticulado de n filas y n columnas que tiene la propiedad de que todas sus filas, y todas sus columnas, y las diagonales principales, suman el mismo valor. Por ejemplo:***

6	***1***	***8***
7	***5***	***3***
2	***9***	***4***

La técnica que se utiliza para generar cuadros mágicos (que tienen siempre una dimensión impar: impar número

de filas y de columnas) es la siguiente:

a. Se comienza fijando el entero 1 en el espacio central de la primera fila.

b. Se van escribiendo los sucesivos números (2, 3, ...) sucesivamente, en las casillas localizadas una fila arriba y una columna a la izquierda. Estos desplazamientos se realizan tratando a la matriz como si estuviera envuelta sobre sí misma, de forma que moverse una posición hacia arriba desde la fila superior lleva a la fila inferior, y moverse una posición a la izquierda desde la primera columna lleva a la columna más a la derecha del cuadro.

c. Si se llega a una posición ya ocupada (es decir, si arriba a la izquierda ya está ocupado con un número anterior), entonces la posición a rellenar cambia, que ahora será la inmediatamente debajo de la última casilla rellenada. Después se continúa el proceso tal y como se ha descrito en el punto anterior.

Escriba un programa que genere el cuadro mágico de la dimensión que el usuario desee, y lo muestre luego por pantalla..

Para llegar a una solución para este programa ofrecemos el flujograma desglosado en partes. Está recogido en la figura 6.1. Con él se puede implementar fácilmente el código que imprima el cuadro mágico. El primer paso (que el usuario introduzca el valor de la dimensión de la matriz cuadrada) debería hacerse de tal manera que sólo se acepta un valor que sea impar; en caso contrario, el programa vuelve a solicitar una dimensión: y así hasta que el usuario acierta a introducir un valor impar.

Aparte del código que cada uno pueda escribir de la mano del flujograma, ofrecemos ahora otro que agiliza de forma notable la búsqueda de la siguiente posición del cuadro donde se ha de colocar el siguiente valor de *numero*. En el código se ha definido una macro mediante la directiva ***#define***. No es trivial verlo a la primera, pero ayuda el ejemplo a comprender que a veces un código bien pensado facilita su comprensión.

El código es el siguiente:

```
#include <stdio.h>
#define lr(x, N) ((x)< 0 ? N+(x)%N : ((x)>=N ? (x)%N : (x) ))

void main(void)
{
      unsigned short magico[17][17];
      unsigned short fil, col, dim, num;

      do
      {
            printf( "\nDimensión ( impar entre 3 y 17 ): ");
            scanf("%hu", &dim);
      }while(dim % 2 == 0);

      printf( "\nCuadro Mágico de dimensión %hu:\n\n", dim);

//Inicializamos la matriz a cero

      for(fil = 0 ; fil < dim ; fil++)
            for(col = 0 ; col < dim ; col++)
                  magico[fil][col] = 0;

// Algoritmo de asignación de valores...

      for(fil = dim/2 , col = 0 , num = 1 ; num < dim*dim;)
      {
            if(magico[fil][col] == 0)
            {
                  magico[fil][col] = num++;
                  fil = lr(fil + 1, dim);
                  col = lr(col - 1, dim);
            }
            else
            {
                  fil = lr(fil - 1, dim);
                  col = lr(col + 2, dim);
            }
      }
```

```
// Mostramos ahora el cuadrado mágico por pantalla

      for(fil = 0 ; fil < dim ; fil++)
      {
            printf("\n\n");
            for(col = 0 ; col < dim ; col++)
                  printf("%5hu", magico[fil][col]);
      }
}
```

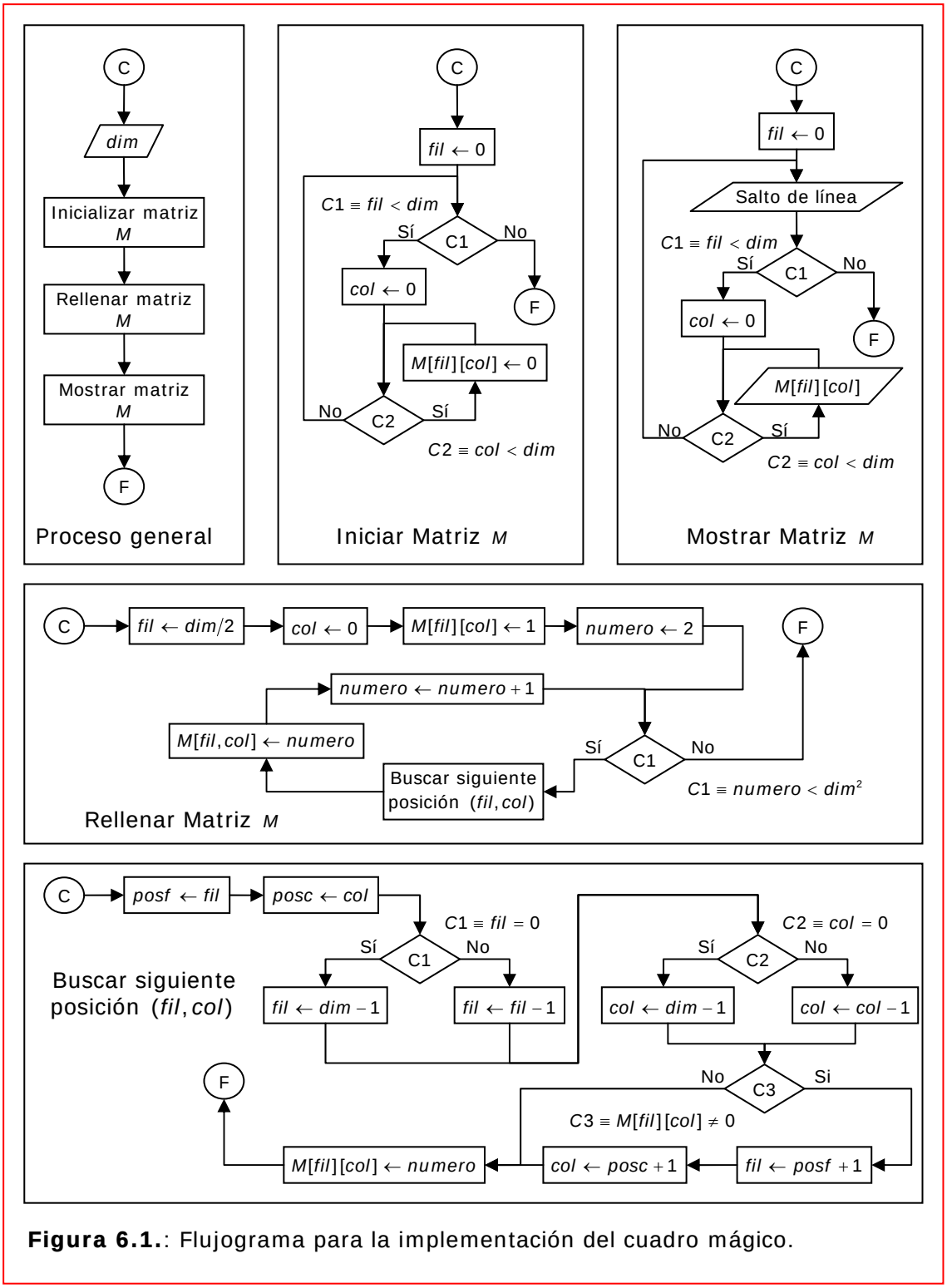

Figura 6.1.: Flujograma para la implementación del cuadro mágico.